A Sparse VLIW Instruction Encoding Scheme Compatible with Generic Binaries

Anthony Brandon*, Joost Hoozemans†, Jeroen van Straten‡, Arthur Lorenzon§, Anderson Sartor¶, Antonio Carlos Schneider Beck‖, Stephan Wong**

Computer Engineering Lab
Delft University of Technology
Email: {a.a.c.brandon*, j.j.hoozemans†, j.s.s.m.wong**}@tudelft.nl, j.vanstraten@student.tudelft.nl‡

Institute of Informatics
Universidade Federal do Rio Grande do Sul
Email: {alorenzon§, alsartor¶, caco‖}@inf.ufrgs.br

***Abstract*—Very Long Instruction Word (VLIW) processors are commonplace in embedded systems due to their inherent low-power consumption as the instruction scheduling is performed by the compiler instead by sophisticated and power-hungry hardware instruction schedulers used in their RISC counterparts. This is achieved by maximizing resource utilization by only targeting a certain application domain. However, when the inherent application ILP (instruction-level parallelism) is low, resources are under-utilized/wasted and the encoding of NOPs results in large code sizes and consequently additional pressure on the memory subsystem to store these NOPs.**

To address the resource-utilization issue, we proposed a dynamic VLIW processor design that can merge unused resources to form additional cores to execute more threads. Therefore, the formation of cores can result in issue widths of 2, 4, and 8. Without sacrificing the possibility of code interruptability and resumption, we proposed a generic binary scheme that allows a single binary to be executed on these different issue-width cores. However, the code size issue remains as the generic binary scheme even slightly further increases the number NOPS.

Therefore, in this paper, we propose to apply a well-known stop-bit code compression technique to the generic binaries that, most importantly, maintains its code compatibility characteristic allowing it to be executed on different cores. In addition, we present the hardware designs to support this technique in our dynamic core. For prototyping purposes, we implemented our design on a Xilinx Virtex-6 FPGA device and executed 14 embedded benchmarks. For comparison, we selected a non-dynamic/static VLIW core that incorporates a similar stop-bit technique for its code compression.

We demonstrate, while maintaining code compatibility on top of a flexible dynamic VLIW processor, that the code size can be significantly reduced (up to 80%) resulting in energy savings, and that the performance can be increased (up to a factor of three). Finally, our experimental results show that we can use smaller caches (2 to 4 times as small), which will further help in decreasing energy consumption.

I. Introduction

VLIW processors exploit ILP by means of a compiler, which statically analyzes the code and builds large instruction words (bundles) composed of instructions (syllables) that will execute in parallel. Since the compiler takes the burden of finding parallelism, VLIW processors occupy less area and dissipate less power when compared to superscalar processors. However, one of the major drawbacks of traditional VLIW processors is the large code size [1]. Because of this, instruction cache misses are more likely for VLIW processors when compared to conventional processors for a given cache size. Consequently, there will be more accesses to the main memory, which has higher delay and needs even more energy when compared to cache memories [2]. One solution would be increasing the cache size. However, this will also significantly increase the power dissipation, as several studies [3], [4] show that the energy consumption of the cache subsystem accounts for over 50% of the overall chip.

One reason for the large code size is the canonical instruction format that dictates the location of the syllables within each instruction word to correspond with the location of the functional units in the different datapaths. Consequently, unused datapaths must be issued NOPs that in turn must be encoded. One way to address this issue is to loosen the relationship between the instruction encoding and the datapath locations at the expense of having a more complex instruction decoder/scheduler. An intermediate solution maintains the canonical nature but encodes all NOPs at the end of an VLIW word with a single bit, the stop bit. These solutions reduce the code size and, consequently, reduce the pressure on the instruction cache (I-cache), reduce the number of I-cache misses, improve the performance of an application, and reduce energy consumption.

The code size issue is mainly due to the low inherent parallelism of the application, i.e., it is not possible to fill all parallel slots of a VLIW word. The nature of VLIW architectures automatically translates this issue into underutilized hardware resources. The ρ-VEX processor was introduced to deal with this low resource utilization by either power gating of datapaths to lower power consumption or merge datapaths together to form additional cores to execute parallel threads (if any). This capability allows the dynamic version of the ρ-VEX to switch at run-time between 2-, 4-, and 8-issue modes. One of the key innovations of the ρ-VEX processor

This work has been supported by the Almarvi European Artemis project nr. 621439.

is the introduction of the generic binary [5] that maintain code compatibility among different (dynamic) configurations of the processor allowing the code to be executable on any configuration (2-, 4-, or 8-issue) and interruptable at any point. The generic binary "suffers" from the same fate as their static counterparts when considering the code size.

In this paper, we propose a modified version of the variable length instruction bundle technique, which can be applied to the generic binaries of the dynamic VLIW processor (ρ-VEX). We also show how this compression technique leverages the current dispatch hardware and extra functional units to make its implementation more straightforward. More specifically, our approach is to apply the sparse instruction encoding (supported by the ISA of the processor) to the dynamic VLIW by using functional units required for supporting dynamic reconfiguration in order to reduce the complexity of dispatching sparse instruction bundles. In conclusion, we are proposing a new approach for code size reduction that marries the benefits of a well-known technique with the dynamic characteristics of the ρ-VEX processor. For comparison purposes, we use a static (non-reconfigurable) VLIW with an implementation of sparse instruction encoding similar to that found in the st200 [1]. The result is twofold: code sizes are reduced, and the difference in performance and energy between the static and dynamic versions decreases (slightly decreasing the price paid for adaptability).

Our contributions in this paper are:

- We implement support for the variable length instruction bundle, based on stop bits, in the dynamic ρ-VEX core to decrease code size, while maintaining compatibility for generic binaries, and therefore the run-time adaptability.
- By leveraging the additional functional units and dispatch logic already present, we show that the extra hardware complexity and overhead needed are insignificant.
- We compare the proposed approach to the non-dynamic version (with and without code compression) of the same processor, which is similar to a processor from the industry (STmicroelectronics' st200 VLIW). We demonstrate that the code compression is highly efficient in both versions, and that in most cases the dynamic version with compression has almost the same performance and energy consumption as the static one.

Comparing the dynamic version with and without the proposed technique, we achieve a code size reduction of over 50%, resulting in cache performance equivalent to that of a cache up to 4 times larger. Additionally, we can save up to 63% on energy consumption, while a maximum speedup of 3 times can be obtained in the best case.

II. Related Work

Conventional VLIW implementations had major drawbacks in the form of low instruction encoding efficiency (a large fraction of the code consisted of NOPs), which, together with the large number of operations that can be needed in a single cycle, resulted in enormous memory bandwidth requirements for instruction fetching. In order to address this issue, several approaches have been proposed:

- *Instruction mask bits*. The MAJC architecture [6] from Sun Microsystems exploits the parallelism at multiple levels: instruction, data, thread and process, through vertical and speculative multithreading, and chip multiprocessing. Mask bits indicate how many and what type of operations the instruction contains. In [7] and [8], the authors use a mask word that encodes which operations are present in the following bundle.
- *Instruction template bits*. These templates are used to limit code size, helping to decode and route the instructions, as used in the Itanium [9] and TM3270 media-processor [10]. The latter uses the templates to determine the compression of the next bundle, which relaxes the timing requirements of the decoding process.
- *Stop-bits*. A bit is reserved in each syllable, indicating whether it is the last syllable in a bundle or not, as presented in [11], [12], [13], [14], and [15].

However, none of the above-mentioned approaches are directly applicable to a dynamically reconfigurable VLIW processor. In order to apply variable length instruction bundle encoding to the ρ-VEX, we implement an extension of the stop-bit approach, which makes it suitable for variable issue widths, therefore maintaining the processor's dynamic adaptability.

III. Implementation

We will compare our implementation of sparse instruction encoding in a dynamic core to an existing encoding scheme applied to a static core. Both schemes are based on the stop-bit approach. The dynamic core is *run-time* reconfigurable in the number of datapaths. It can be configured as an 8-, 4-, or 2-issue core, whereas the static core has a fixed issue-width of 4 because the st200 toolchain which we use to compile applications for it only supports 4-issue.

A. Shared requirements

A number of properties are required by both cores. They will be outlined here and the precise implementations will be discussed in their respective sections. To support a sparse encoding, the hardware must support the following:

- Instruction bundles of variable length. That is, the location of the stop-bit determines which syllables should be executed, and also impacts the calculation of the next Program Counter (NextPC).
- Instruction bundles that cross a cache line boundary. An instruction buffer is used to store the relevant parts of the previous cache line to accommodate this in both designs.

B. Static Core

1) Overview: The VEX ISA is very closely related to STmicroelectronics' st200/Lx [11]. Both use an instruction encoding scheme with the following restrictions [16] that help to reduce the complexity of the fetch hardware [17]:

- Branch operations must be the first syllable in a bundle.

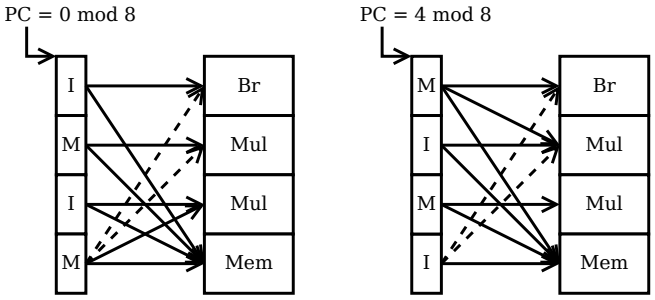

Fig. 1. Possibilities for the dynamic issue hardware using the st200 encoding scheme for bundles stored on even (left) and odd (right) word addresses.

- Multiplication operations must be stored at odd word addresses. This also restricts the number of multiplication operations to two per bundle.
- Long immediate extensions must be stored at even word addresses.

Considering these restrictions, the next section discusses the hardware modifications necessary for the implementation of the encoding technique.

2) Hardware: As opposed to the st200 implementation, our instruction cache does not support unaligned accesses and the core is designed to be generic (design-time configurable). To be generic, the design is very "lane-oriented", because the number of execution lanes or "datapaths" is generic, and every lane can be configured with multiple functional units (load/store, branch, ALU, and multiplication unit).

The first step in order to enable variable sized instruction bundles in the design is to add support for unaligned instruction cache accesses and bundles that cross a cache-line boundary. To this end, we add an instruction buffer between the cache and the fetch unit. This adds an additional stage to the datapath, increasing the branch delay by one cycle. The instruction buffer is similar to the design discussed in Section III-C2a.

The second step is to add dispatching logic that can send each syllable to a datapath that contains the functional unit that can execute the syllables operation type. The hardware required to fully support this can be quite complex. When every datapath needs to be able to accept an operation from any syllable slot in a bundle, a full crossbar is required [14]. Fortunately, the restrictions in the encoding scheme reduce this routing complexity to the diagram depicted in Fig. 1. The figure shows the locations of the MEM (load/store), MUL (multiplication) and BR (branch) units. Each datapath also contains an ALU, which is not depicted. The figure also depicts whether each syllable slot can contain a long immediate extension (I) or a multiplication (M) for even or odd instruction bundle start addresses. The arrows show which lanes it is possible to dispatch a particular syllable to. The dotted arrows depict an indirect requirement that is needed when two operations need to be swapped (e.g., if a MEM operation is located in syllable 0 and an ALU operation is located in slot 3, they will need to be swapped as the MEM unit is located in datapath 3).

TABLE I
LAYOUT OF FUNCTIONAL UNITS IN AN 4-ISSUE DYNAMIC ρ-VEX.

Datapath 0	Datapath 1	Datapath 2	Datapath 3
ALU	ALU	ALU	ALU
MUL	MUL	MUL	MUL
MEM	BR	MEM	BR

3) Discussion: As we can see in Fig. 1 the dispatch logic for a 4-issue core is already quite complex. Expanding the dispatch hardware discussed in this section to an 8-way VLIW would complicate the circuitry even more. The complexity would increase considerably, but not exponentially, because not all of the functional units (e.g. load/store, branch) are duplicated when doubling the issue width. As we will see in the following sections, our approach in the dynamic ρ-VEX core is able to dispatch operations with simpler logic that can be more efficiently scaled to an 8-way VLIW.

C. Dynamic Core

1) Overview: The dynamic core consists of multiple datapaths, which are divided into groups of two. Each datapath has a fixed set of functional units. Each group of two datapaths can function as a separate VLIW core, or can be combined with adjacent groups to form a larger VLIW core. In order to allow this, each group of two datapaths must have a functional unit layout that is identical to that of the the other groups. Additionally, to allow each datapah group to function as a separate VLIW core, each group must have a load/store unit and a branch unit. However, when the groups are configured to combine into a 4-issue VLIW, these additional load/store and branch units go unused. For example, the layout of functional units in each datapath is shown in Table I for a 4-issue configuration. By using this arrangement of functional units we are able to support sparse instruction encoding without decoding logic to dispatch instructions to different functional units.

2) Hardware: In addition to the requirements mentioned in Section III-A, in order to support sparse instruction encoding in the dynamic core, we also require the next PC calculation to be able to support dynamic switching of issue-width. This means that it should be possible to calculate either a single address, or multiple addresses based on the core configuration. Additionally, unlike in the static core, branch instructions must be able to appear in any lane to maintain the single/multiple core adaptability.

a) Instruction Buffer: When using sparse instruction encoding, bundles no longer have a fixed size. Because of this, the cache line size is no longer divisible by the bundle size, which means that instruction bundles can cross cache line boundaries. We handle this by fetching the next cache line and storing the previous one in a buffer in order to complete the instruction bundle.

Fig. 2 shows a diagram of the instruction buffer. Each lane-group has a register to store the previously fetched syllables

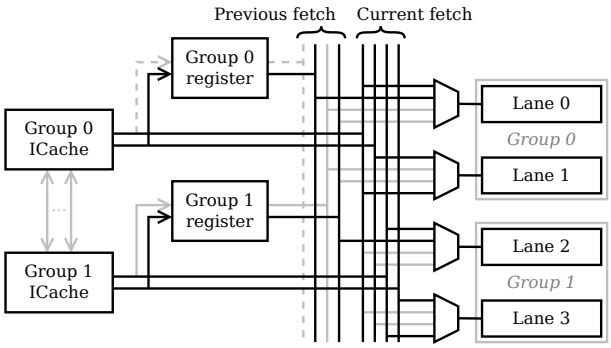

Fig. 2. This figure shows how the instruction buffer is implemented for a 4-issue dynamic core consisting of two lane groups.

(two syllables per group). The muxes select which syllable goes to which datapath based on the current configuration of the core (2-, 4-, or 8-issue), and the least significant bits of the program counter. The gray lines indicate paths that are only used when the core is configured in 4-issue mode. The dotted line indicates a path which would be used if the core were in 8-issue mode.

The instruction buffer registers are loaded whenever a new fetch address is sent to the cache. This can be determined by comparing the least significant bit of the current and previous fetch address, reducing the size of the required comparator significantly. Note that when a branch to an unaligned address occurs, the syllables in the instruction buffer are undefined. Additional logic is present in the branch unit to stall the core for an extra cycle in this case, during which an additional instruction fetch is performed in order to fill the instruction. This additional stall could adversely affect performance if it occurs often enough.

The muxes that select between the previously fetched data and the current fetched data are controlled by signals based on the current configuration (2-issue, 4-issue, or 8-issue) and the LSBs (Least Significant Bits) of the Program Counter (PC). For the case where the core is configured as the largest possible configuration (all lane-groups work together as one core) the mux select signals are equal to the least significant bits of the Program Counter.

b) Address Calculation: When using variable sized instruction bundles, the next value of the program counter depends on the size of the current fetched instruction bundle. This complicates the calculation of the next PC, which can now be PC + 4, 8, 12 or 16 (for a 4-issue VLIW). We deal with this by splitting the calculations into two parts:

- Calculation of the least significant bits (depicted in Fig. 3) is done for each lane to determine what the least significant bits would be if the instruction bundle ended in a particular lane. In each lane, a value corresponding to a different instruction bundle size is added to the least significant bits of the PC. For example, if the stop-bit is in the first instruction, that means the instruction bundle size is 4 bytes, and 4 is added to the PC. Additionally, the "align up" adder rounds up the calculated least significant bits to a cache line boundary. This will be used for the next fetch address.
- The most significant bits of the next PC are calculated in each branch unit (for the 4-issue configuration used in this paper that would be lanes 1 and 3). Finally, the least and most significant bits are combined to form the next PC based on the final position of the stop bit.

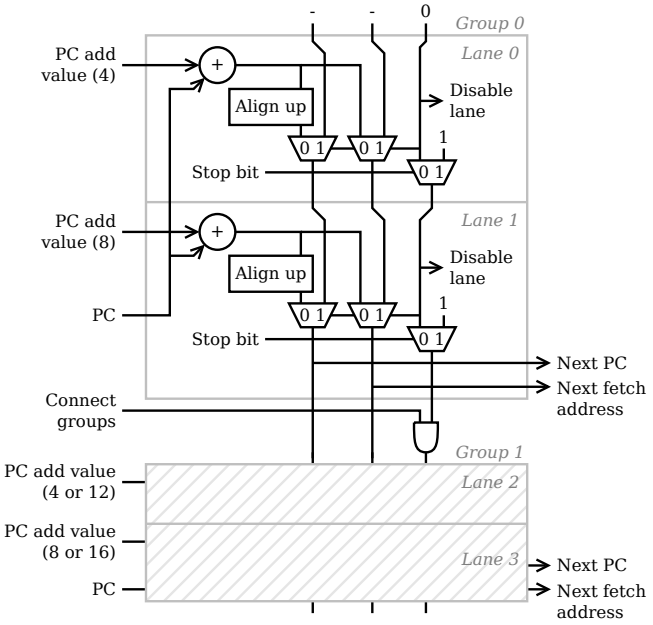

Fig. 3. This figure shows how the LSB of the next PC and next fetch address are calculated in each lane.

By splitting the program counter calculation in this way, we only need four 27-bit adders to calculate the most significant bits, of which only two are used in 4-way mode, and eight 3-bit adders for the least significant bits, instead of needing eight 32-bit adders.

c) Branch Instruction Dispatch: As mentioned in [5], generic binaries require that the branch instruction is always the last in a bundle, rather than the first. This requirement is in place to ensure that the bundle will still be executed completely by a core running in a 2-way configuration (otherwise, the syllables following the branch would be skipped because of the branch). In order to support this, additional logic is present to route the last instruction in a bundle to the last coupled lane if it is a branch instruction, so only one physical branch resource is used for executing branch instructions.

Because instruction bundles can now cross cache line boundaries, it is possible that after a branch only part of the next instruction bundle is fetched. The hardware detects this by checking if one of the fetched syllables (starting from the branch target address) has a stop-bit set. If not, the core stalls while the second part of the bundle is fetched. This means that for unaligned branches the branch delay is two cycles instead of one, which can cause performance degradation.

TABLE II
THE RESOURCE USAGE ON THE FPGA FOR THE DYNAMIC CORE WITH AND WITHOUT STOP-BIT IMPLEMENTATION.

Resource	Original	Stop-bit	Increase
Registers	30153	30537	1.3%
Luts	61927	62379	0.7%
BRAMs	125	125	0.0%

IV. RESULTS

We evaluated four different versions of the processors: static baseline, static with stop-bit, dynamic baseline, and dynamic with stop-bit — all of them in their 4-issue configurations. We use the 4-issue configurations to provide a fair comparison between the static and dynamic cores. The difference between dynamic and static versions is that the binaries for the former are compiled as generic binaries. We considered instruction cache sizes ranging from 1KiB to 32KiB. These sizes were chosen so that at the largest cache size each of the programs fits in the instruction cache entirely.

The designs are implemented in VHDL and prototyped on a Xilinx Virtex 6 FPGA (ML605 Development board). With these prototypes, we use performance counters to determine the number of cache accesses, misses, and the number of running cycles. The cache stall time is 16 cycles per 4-byte bus access. We use the Cadence Encounter RTL Compiler to obtain power dissipation in ASIC (Application Specific Integrated Circuit), using a 65nm CMOS cell library from STMicroeletronics. The energy consumption of the memory subsystem was calculated with the Cacti Tool [18].

We use applications from the Powerstone benchmarks [19]. All sources are compiled with the HP VEX compiler [20] and assembled with either the ρ-VEX port of GNU as, or our modified version of the st200 assembler. The dynamic stop-bit versions are assembled with alignment turned off, so that instruction bundles are not padded at all. Since the processor lacks floating point operations, we use the floatlib library included with the HP VEX compiler (based on Berkeley SoftFloat [21]).

A. FPGA Resource usage

Table II shows the resource usage of the dynamic core on the FPGA. It shows that the increase is only 1.3% for the number of registers and 0.7% for the number of lookup tables. As we will show in the following sections, with this small increase in area we achieve significant improvements in performance, energy, and code size.

B. Code Size Reduction and Instruction Cache Miss Rate

In Table III, we show the reduction in code size for each of the 14 benchmarks used. We can see that the average reduction is around 50%. The reductions for the dynamic core in 8-way configuration are included for reference, and are even more extreme. These reductions will impact the cache behavior. In Fig. 4, we show the cache miss rates for the two different cores with and without sparse instruction encoding. The results

TABLE III
THE CODE SIZE REDUCTION FOR EACH OF THE BENCHMARKS.

Program	code size reduction		
	static 4-way core	dynamic 4-way core	dynamic 8-way core
adpcm	49%	48%	73%
bcnt	35%	38%	64%
blit	47%	45%	67%
compress	53%	51%	74%
crc	48%	48%	71%
des	42%	44%	68%
engine	57%	54%	77%
fir	60%	54%	76%
g3fax	58%	55%	76%
jpeg	53%	51%	73%
pocsag	55%	51%	74%
qurt	67%	65%	82%
ucbqsort	57%	54%	76%
v42	56%	53%	75%
average	53%	51%	73%

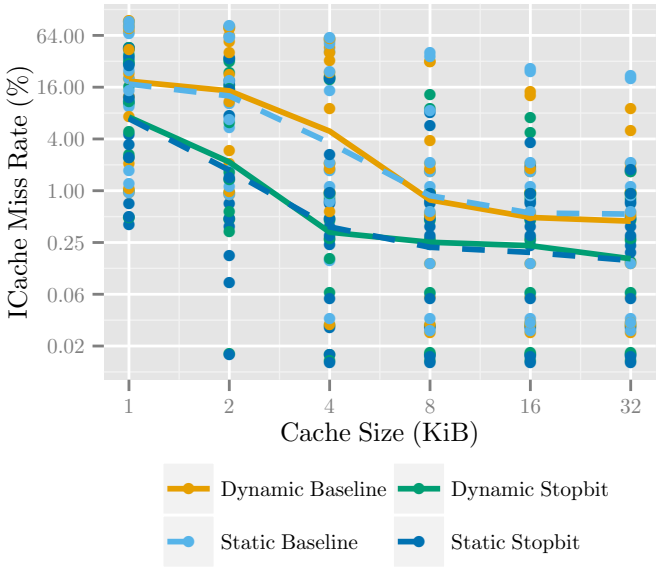

Fig. 4. Cache miss percentage for the dynamic and static cores, both with and without sparse instruction encoding for different instruction cache sizes. The dots represent the individual benchmarks, whereas the lines represent the average miss percentage for a particular configuration.

show that both designs achieve a similar reduction in cache misses. In fact, with sparse instruction encoding the miss rates are similar to those of canonical encoding with a cache almost four times as large. This might seem like a larger improvement than expected, since the code size was only reduced by half. However, because loops account for a majority of the executed instructions, code size reduction that allow an entire loop body to fit into the cache will have a disproportionate impact on the cache miss rate.

C. Execution Time

Fig. 5 shows the speedup in execution time achieved for both the dynamic and static cores. We can see that for larger cache sizes, the execution time of some benchmarks is larger

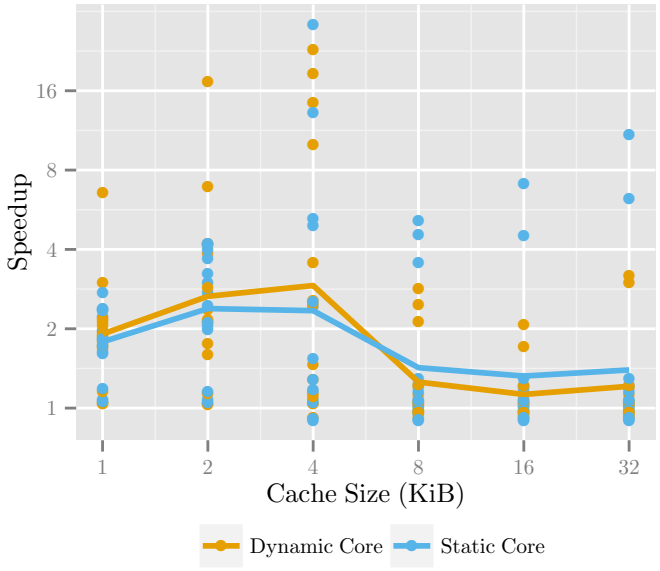

Fig. 5. Speedup for stop-bit implementation for different instruction cache sizes. The lines represent the average speedup for a particular cache size.

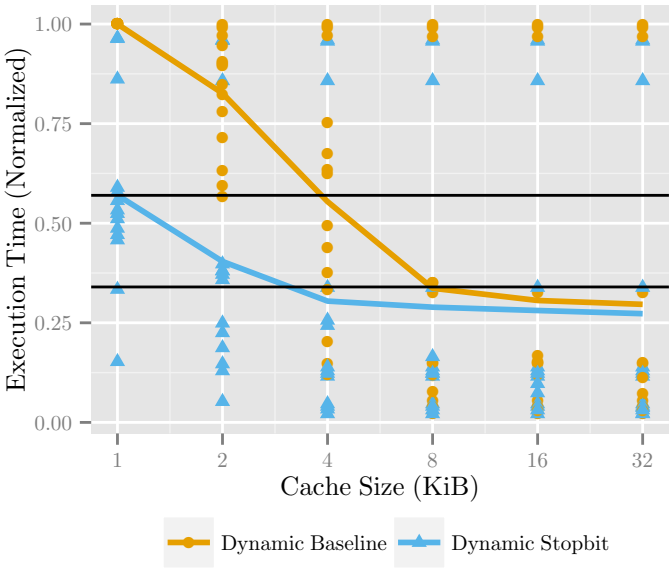

Fig. 6. Normalized execution times for the dynamic core

with sparse instruction encoding than without. This is because at those cache sizes the entire application fits in the cache, and the reduction in cache misses is offset by the penalty of having a longer branch delay. This could be remedied by inserting alignment NOPs to ensure that branch targets are always aligned, at the cost of an increase in cache misses.

In the same figure we also observe that for very small instruction cache sizes, the speedup is not as significant as it is for intermediate sizes. This is caused by the fact that the reduction in cache misses for intermediate cache sizes is far larger than for small cache sizes, as seen in Fig. 4.

Fig. 6 shows the normalized execution times for the dynamic core. The baseline is the average of the worst execution time of each application individually, executing on the dynamic baseline design with a cache size of 1KiB. This figure shows, for instance, that for smaller cache sizes, the dynamic stop-bit implementation performs equivalent to the dynamic version without stop-bit with a cache between 2 and 4 times larger.

D. Energy Results

Fig. 7 presents the total energy consumed by each of the benchmarks. The lines show the geometric mean of all applications at each cache size. We can see that for small cache sizes, due to the additional hardware required to support reconfiguration, the dynamic core consumes more energy than the static core. However, at large cache sizes the different designs are closer together in terms of energy consumption.

Fig. 8 depicts the energy consumption of the dynamic core relative to that of the static core (values greater than 1 mean that the static version consumes less energy than the dynamic one). We can see that the baseline dynamic design consumes far more energy at small cache sizes, whereas when

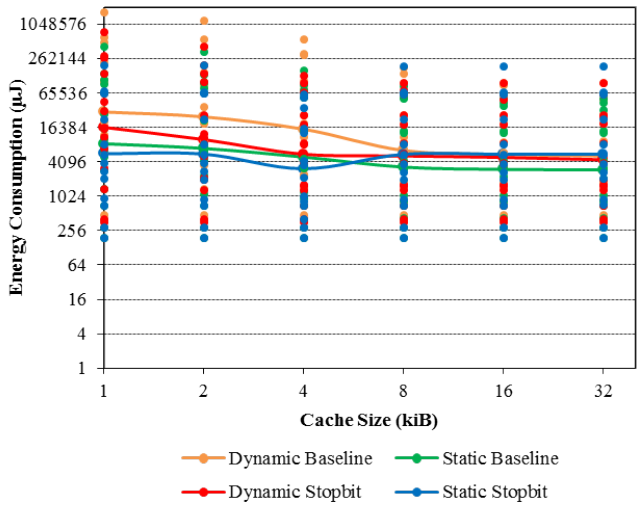

Fig. 7. Energy consumption for each of the benchmarks at different cache sizes.

using sparse instruction encoding the designs consume similar amounts of energy.

As one can observe, the huge difference in energy consumption between the static and dynamic versions is significantly decreased when using the proposed stop-bit approach. Most notably, both processors consume approximately the same amount of energy at larger cache sizes. It means that one can take advantage of all the adaptability that the dynamic version provides, with limited additional costs in terms of energy.

V. Conclusion

In this paper, we extended the stop-bit technique for sparse instruction encoding to a dynamically reconfigurable VLIW processor. We showed that, by implementing this technique,

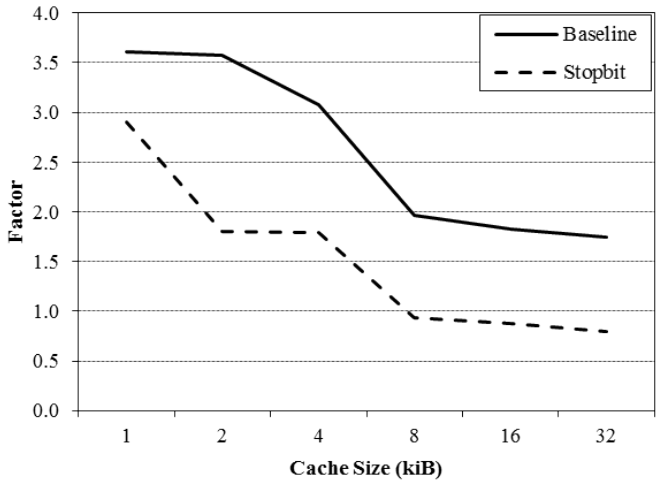

Fig. 8. Relative energy consumption between the static and dynamic cores (dynamic/static) at different cache sizes.

we reduce the cost of reconfigurability in terms of energy consumption and the performance overhead of cache misses. This is achieved without sacrificing the code compatibility of the generic binary and we thereby maintain full (dynamic) adaptability of the core. Using this technique, we bring the energy consumption of the dynamic core closer to that of the static design. Our results show that using the stop-bit technique in the dynamic core we can achieve similar performance and energy consumption with up to $4\times$ smaller I-caches.

We do notice that for some applications at certain cache sizes the performance with stop-bit is slightly lower than without stop-bit due to the increased branch delay. Therefore, for future work we will investigate the effect of ensuring that branch target addresses are always correctly aligned. This would result in a slight increase in cache misses but also a decrease in delays due to branches.

REFERENCES

[1] P. Faraboschi, G. Brown, J. Fisher, G. Desoll, and F. Homewood, "Lx: a technology platform for customizable vliw embedded processing," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, June 2000, pp. 203–213.

[2] V. A. Korthikanti and G. Agha, "Towards optimizing energy costs of algorithms for shared memory architectures," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 157–165. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810510

[3] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache for low energy embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 2, pp. 363–387, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1067915.1067921

[4] W. Wang, P. Mishra, and S. Ranka, "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 948–953. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024935

[5] A. Brandon and S. Wong, "Support for dynamic issue width in vliw processors using generic binaries," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 827–832.

[6] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse, "The majc architecture: A synthesis of parallelism and scalability," *IEEE Micro*, vol. 20, no. 6, pp. 12–25, 2000.

[7] Instruction storage method with a compressed format using a mask word, www.google.com/patents/US5057837.

[8] S. Jee and K. Palaniappan, "Performance evaluation for a compressed-vliw processor," in *Proceedings of the 2002 ACM symposium on Applied computing*. ACM, 2002, pp. 913–917.

[9] H. Sharangpani and K. Arora, "Itanium Processor Microarchitecture," *IEEE Micro*, pp. 24–43, Sep. 2000.

[10] J.-W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra *et al.*, "The TM3270 media-processor," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2005, pp. 331–342.

[11] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. 500 Sansome Street, Suite 400, San Francisco, CA 94111: Morgan Kaufmann Publishers, 2005.

[12] Method and apparatus for sequencing and decoding variable length instructions with an instruction boundary marker within each instruction, http://www.google.com/patents/US5881260.

[13] A. Suga and K. Matsunami, "Introducing the fr500 embedded microprocessor," *Micro, IEEE*, vol. 20, no. 4, pp. 21–27, 2000.

[14] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," in *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*. IEEE, 1996, pp. 201–211.

[15] B. Hubener, G. Sievers, T. Jungeblut, M. Porrmann, and U. Ruckert, "Coreva: A configurable resource-efficient vliw processor architecture," in *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*. IEEE, 2014, pp. 9–16.

[16] ST231 Core and Instruction Set Architecture Manual.

[17] Instruction fetch apparatus for wide issue processors and method of operation, http://www.google.com/patents/US7028164.

[18] S. Thoziyoor, J. H. Ahn, A. Monchiero, J. B. Brockman, and N. P. Jouppi, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," in *Proc. 35th International Symposium on Computer Architecture (35th ISCA'08)*. Beijing: ACM SIGARCH, Jun. 2008.

[19] Powerstone Benchmarks, http://www.cprover.org/goto-cc/examples/index.php.

[20] The HP VEX toolchain, http://www.hpl.hp.com/downloads/vex/.

[21] http://www.jhauser.us/arithmetic/SoftFloat.html.